MODULE 4

Syllabus:
Packages: Packages, Access Protection, Importing Packages
Interfaces
Exception Handling: Exception-Handling Fundamentals, Exception Types, Uncaught Exceptions, Using try and catch, Multiple catch Clauses, Nested try Statements, throw, throws, finally, Java's Built-in Exceptions, Creating Your Own Exception Subclasses, Chained Exceptions, Using Exceptions.

4.1 Packages

When we have more than one class in our program, usually we give unique names to classes. In a real-time development, as the number of classes increases, giving unique meaningful name for each class will be a problem. To avoid name-collision in such situations, Java provides a concept of packages. **A package is a collection of classes.** The package is both a naming and a visibility control mechanism. You can define classes inside a package that are not accessible by code outside that package. You can also define class members that are only exposed to other members of the same package. This allows your classes to have intimate knowledge of each other, but not expose that knowledge to the rest of the world.

4.1.1 Defining a Package

To create a package, include a ***package*** command as the first statement in a Java source file. Any class declared within that file will belong to the specified package. If you omit the package statement, the class names are put into the default package, which has no name.

General form of the package statement:

package *pkg;*

Example –

```
package MyPackage;
```

Java uses file system directories to store packages. For example, the `.class` file for any class you declare to be part of `MyPackage` must be stored in a directory called `MyPackage`. Remember that case is significant, and the directory name must match the package name exactly. More than one file can include the same package statement. The package statement simply specifies to which package the classes defined in a file belong. It does not exclude other class in other files from being part of that same package.

One can create a hierarchy of packages. To do so, simply separate each package name from the one above it by use of a period. The general form of a multileveled package statement is shown here:

```
package pkg1[.pkg2[.pkg3]];
```

A package hierarchy must be reflected in the file system of your Java development system. For example, a package declared as package `java.awt.image`; needs to be stored in `java\awt\image` in a Windows environment. You cannot rename a package without renaming the directory in which the classes are stored.

4.1.2 Finding Packages and CLASSPATH

As we have seen, packages are reflected with directories. This will raise the question that - how does Java run-time know where to look for the packages that we create?

- By default, Java run-time uses current working directory as a starting point. So, if our package is in a sub-directory of current working directory, then it will be found.
- We can set directory path using CLASSPATH environment variable.
- We can use **–classpath** option with **javac** and **java** to specify path of our classes.

Assume that we have created a package ***MyPackage.*** When the second two options are used, the class path ***must not*** include **MyPackage**. It must simply specify the *path to* **MyPackage**. For example, in a Windows environment, if the path to **MyPackage** is

C:\MyPrograms\Java\MyPackage

Then the class path to **MyPackage** is

C:\MyPrograms\Java

Consider the program given below –

```
package MyPackage;

class Test
{
     int a, b;
     Test(int x, int y)
     {
          a=x; b=y;
     }

     void disp()
     {
          System.out.println("a= "+a+" b= "+b);
     }
}

class PackDemo
{
     public static void main(String args[])
     {
          Test t=new Test(2,3);
          t.disp();
     }
}
```

4.1.3 Access Protection

Java provides many levels of protection to allow fine-grained control over the visibility of variables and methods within classes, subclasses, and packages. Classes and packages are both means of encapsulating and containing the name space and scope of variables and methods. Packages act as containers for classes and other subordinate packages. Classes act as containers for data and code. The class is Java's smallest unit of abstraction.

Java addresses four categories of visibility for class members:

- Subclasses in the same package
- Non-subclasses in the same package

- Subclasses in different packages
- Classes that are neither in the same package nor subclasses

Even a class has accessibility feature. A class can be kept as default or can be declared as ***public.*** When a class is declared as **public**, it is accessible by any other code. If a class has default access, then it can only be accessed by other code within its same package. When a class is public, it must be the only public class declared in the file, and the file must have the same name as the class

Accessibility of members of the class can be better understood using the following table.

	Private	**No Modifier**	**Protected**	**Public**
Same class	Yes	Yes	Yes	Yes
Same package subclass	No	Yes	Yes	Yes
Same package non-subclass	No	Yes	Yes	Yes
Different package Subclass	No	No	Yes	Yes
Different package non-subclass	No	No	No	Yes

4.1.4 Importing Packages

Since classes within packages must be fully qualified with their package name or names, it could become tedious to type in the long dot-separated package path name for every class you want to use. For this reason, Java includes the **import** statement to bring certain classes, or entire packages, into visibility. Once imported, a class can be referred to directly, using only its name.

In a Java source file, **import** statements occur immediately following the **package** statement (if it exists) and before any class definitions. The general form of the **import** statement is:

```
import pkg1[.pkg2].(classname|*);
```

For example,

```
import java.util.Date;
import java.io.*;
```

The star form may increase compilation time—especially if you import several large packages. For this reason it is a good idea to explicitly name the classes that you want to use rather than importing whole packages. However, the star form has absolutely no effect on the run-time performance or size of your classes.

All of the standard Java classes included with Java are stored in a package called **java**. The basic language functions are stored in a package inside of the **java** package called **java.lang**. Normally, you have to import every package or class that you want to use, but since Java is useless without much of the functionality in **java.lang**, it is implicitly imported by the compiler for all programs. This is equivalent to the following line being at the top of all of your programs:

```
import java.lang.*;
```

If a class with the same name exists in two different packages that you import using the star form, the compiler will remain silent, unless you try to use one of the classes. In that case, you will get a compile-time error and have to explicitly name the class specifying its package.

The **import** statement is optional. Any place you use a class name, you can use its *fully qualified name*, which includes its full package hierarchy. For example,

```
import java.util.*;

class MyDate extends Date
{ ……………}
```

Can be written as –

```
class MyDate extends java.util.Date
{ …}
```

4.2 Interfaces

Interface is an abstract type that can contain only the declarations of methods and constants. Interfaces are syntactically similar to classes, but they do not contain instance variables, and their methods are declared without any body. Any number of classes can ***implement*** an interface. One class may implement many interfaces. By providing the **interface** keyword, Java allows you to fully utilize the "one interface, multiple methods" aspect of polymorphism. Interfaces are alternative means for multiple inheritance in Java.

4.2.1 Defining an Interface

An interface is defined much like a class. This is the general form of an interface:

```
access interface name
{
     type final-varname1 = value;
     type final-varname2 = value;
     …………………
     return-type method-name1(parameter-list);
     return-type method-name2(parameter-list);
     …………………
}
```

Few key-points about interface:

- When no access specifier is mentioned for an interface, then it is treated as *default* and the interface is only available to other members of the package in which it is declared. When an interface is declared as **public**, the interface can be used by any other code.
- All the methods declared are abstract methods and hence are not defined inside interface. But, a class implementing an interface should define all the methods declared inside the interface.
- Variables declared inside of interface are implicitly **final** and **static**, meaning they cannot be changed by the implementing class.
- All the variables declared inside the interface must be initialized.
- All methods and variables are implicitly **public**.

4.2.2 Implementing Interface

To implement an interface, include the ***implements*** clause in a class definition, and then create the methods defined by the interface. The general form of a class that includes the implements clause looks like this:

```
class classname extends superclass implements interface1, interface2...
{
     // class-body
}
```

Consider the following example:

```
interface ICallback
{
     void callback(int param);
}

class Client implements ICallback
{
     public void callback(int p)           //note public
     {
          System.out.println("callback called with " + p);
     }
     void test()
     {
          System.out.println("ordinary method");
     }
}

class TestIface
{
     public static void main(String args[])
     {
          ICallback c = new Client();
          c.callback(42);

          //   c.test()              //error!!
     }
}
```

Here, the interface *ICallback* contains declaration of one method *callback()*. The class *Client* implementing this interface is defining the method declared in interface. Note that, the method *callback()* is public by default inside the interface. But, the keyword *public* must be used while defining it inside the class. Also, the class has its own method *test()*. In the main() method, we are creating a reference of interface pointing to object of *Client* class. Through this reference, we can call interface method, but not method of the class.

The true polymorphic nature of interfaces can be found from the following example –

```
interface ICallback
{
     void callback(int param);
}

class Client implements ICallback
{
     public void callback(int p)          //note public
     {
          System.out.println("callback called with " + p);
     }
}

class Client2 implements ICallback
{
     public void callback(int p)
     {
          System.out.println("Another version of ICallBack");
          System.out.println("p squared " + p*p);
     }
}

class TestIface
{
     public static void main(String args[])
     {
          ICallback x[]={new Client(), new Client2()};

          for(int i=0;i<2;i++)
               x[i].callback(5);
     }
}
```

Output:

```
callback called with 5
Another version of ICallBack
p squared 25
```

In this program, we have created array of references to interface, but they are initialized to class objects. Using the array index, we call respective implementation of *callback()* method.

Note: Interfaces may look similar to abstract classes. But, there are lot of differences between them as shown in the following table:

Abstract Class	Interface
Can have instance methods that implements a default behavior.	Are implicitly abstract and cannot have implementations.
May contain non-final variables.	Variables declared in interface are by default final.

Can have the members with private, protected, etc..	Members of a Java interface are public by default.
A Java abstract class should be extended using keyword "extends".	Java interface should be implemented using keyword "implements"
An abstract class can extend another Java class and implement multiple Java interfaces.	An interface can extend another Java interface only.
Not slow	Compared to abstract classes, interfaces are slow as it requires extra indirection.

4.2.3 Variables in Interfaces

You can use interfaces to import shared constants into multiple classes by simply declaring an interface that contains variables that are initialized to the desired values. When you include that interface in a class all of those variable names will be in scope as constants (Similar to #define in C/C++). If an interface contains no methods, then any class that includes such an interface doesn't actually implement anything. It is just using a set of constants. Consider an example to illustrate the same:

```
interface SharedConst
{
     int FAIL=0;          //these are final by default
     int PASS=1;
}

class Result implements SharedConst
{
     double mr;

     Result(double m)
     {
          mr=m;
     }

     int res()
     {
          if(mr<40)
               return FAIL;
          else return PASS;
     }
}
class Exam extends Result implements  SharedConst
{
     Exam(double m)
     {
          super(m);
     }
```

```
        public static void main(String args[])
        {
             Exam r = new Exam(56);

             switch(r.res())
             {
                  case FAIL:
                       System.out.println("Fail");
                       break;
                  case PASS:
                       System.out.println("Pass");
                       break;
             }
        }
   }
```

4.2.4 Interfaces can be extended

One interface can inherit another interface by using the keyword **extends**. The syntax is the same as for inheriting classes. When a class implements an interface that inherits another interface, it must provide implementations for all methods defined within the interface inheritance chain.

```
interface A
{
   void meth1();
   void meth2();
}

interface B extends A
{
   void meth3();
}

class MyClass implements B
{
   public void meth1()
   {
        System.out.println("Implement meth1().");
   }

   public void meth2()
   {
        System.out.println("Implement meth2().");
   }
   public void meth3()
   {
        System.out.println("Implement meth3().");
   }
}
```

```
class IFExtend
{
   public static void main(String arg[])
   {
         MyClass ob = new MyClass();
         ob.meth1();
         ob.meth2();
         ob.meth3();
   }
}
```

4.3 Exception Handling

An ***exception*** is an abnormal condition that arises in a code sequence at run time. In other words, an exception is a run-time error. In computer languages that do not support exception handling, errors must be checked and handled manually—typically through the use of error codes. This approach is as cumbersome as it is troublesome. Java's exception handling avoids these problems and, in the process, brings run-time error management into the object oriented world.

4.3.1 Exception Handling Fundamentals

A Java exception is an object that describes an exceptional (that is, error) condition that has occurred in a piece of code. When an exceptional condition arises, an object representing that exception is created and ***thrown*** in the method that caused the error. That method may choose to handle the exception itself, or pass it on. Either way, at some point, the exception is ***caught*** and processed.

Exceptions can be generated by the Java run-time system, or they can be manually generated by your code. Exceptions thrown by Java relate to fundamental errors that violate the rules of the Java language or the constraints of the Java execution environment. Manually generated exceptions are typically used to report some error condition to the caller of a method.

Java exception handling is managed using five keywords:

- **try**: A suspected code segment is kept inside ***try*** block.
- **catch:** The remedy is written within ***catch*** block.
- **throw:** Whenever run-time error occurs, the code must ***throw*** an exception.
- **throws:** If a method cannot handle any exception by its own and some subsequent methods needs to handle them, then a method can be specified with ***throws*** keyword with its declaration.
- **finally:** block should contain the code to be executed after finishing try-block.

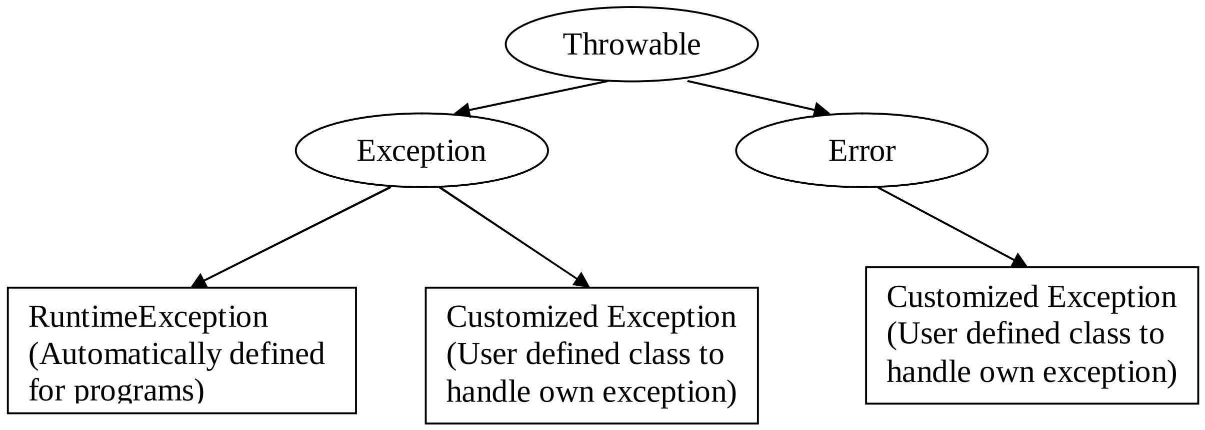
The general form of exception handling is –

```
try
{
      // block of code to monitor errors
}
catch (ExceptionType1 exOb)
{
      // exception handler for ExceptionType1
}
catch (ExceptionType2 exOb)
{
```

```
            // exception handler for ExceptionType2
        }
        ...
        ….
        finally
        {
            // block of code to be executed after try block ends
        }
```

4.3.2 Exception Types

All the exceptions are the derived classes of built-in class viz. ***Throwable.*** It has two subclasses viz. ***Exception*** and ***Error.***


Throwable
Exception
Error
RuntimeException (Automatically defined for programs)
Customized Exception (User defined class to handle own exception)
Customized Exception (User defined class to handle own exception)

Exception class is used for exceptional conditions that user programs should catch. We can inherit from this class to create our own custom exception types. There is an important subclass of **Exception**, called **RuntimeException**. Exceptions of this type are automatically defined for the programs that you write and include things such as division by zero and invalid array indexing.

Error class defines exceptions that are not expected to be caught under normal circumstances by our program. Exceptions of type **Error** are used by the Java run-time system to indicate errors having to do with the run-time environment, itself. Stack overflow is an example of such an error.

4.3.3 Uncaught Exceptions

Let us see, what happens if we do not handle exceptions.

```
        class Exc0
        {
            public static void main(String args[])
            {
                int d = 0;
                int a = 42 / d;
            }
        }
```

When the Java run-time system detects the attempt to divide by zero, it constructs a new exception object and then *throws* this exception. This causes the execution of Exc0 to stop, because once an exception has been thrown, it must be *caught* by an exception handler and dealt with immediately. Since, in the above program, we have not supplied any exception handlers of our own, so the exception is caught by the default handler provided by the Java run-time system.

Any un-caught exception is handled by default handler. The default handler displays a string describing the exception, prints a stack trace from the point at which the exception occurred, and terminates the program. Here is the exception generated when above example is executed:

```
java.lang.ArithmeticException: / by zero
at Exc0.main(Exc0.java:6)
```

The stack trace displays *class name, method name, file name* and *line number* causing the exception. Also, the type of exception thrown viz. *ArithmeticException* which is the subclass of *Exception* is displayed. The type of exception gives more information about what type of error has occurred. The stack trace will always show the sequence of method invocations that led up to the error.

```
class Exc1
{
	static void subroutine()
	{
		int d = 0;
		int a = 10 / d;
	}
	public static void main(String args[])
	{
		Exc1.subroutine();
	}
}
```

The resulting stack trace from the default exception handler shows how the entire call stack is displayed:

```
java.lang.ArithmeticException: / by zero
at Exc1.subroutine(Exc1.java:6)
at Exc1.main(Exc1.java:10)
```

4.3.4 Using try and catch

Handling the exception by our own is very much essential as

- We can display appropriate error message instead of allowing Java run-time to display stack-trace.
- It prevents the program from automatic (or abnormal) termination.

To handle run-time error, we need to enclose the suspected code within *try* block.

```
class Exc2
{
	public static void main(String args[])
	{
		int d, a;

		try
		{
			d = 0;
```

```
                    a = 42 / d;
                    System.out.println("This will not be printed.");
               } catch (ArithmeticException e)
               {
                    System.out.println("Division by zero.");
               }
               System.out.println("After catch statement.");
          }
     }
```

Output:

```
     Division by zero.
     After catch statement.
```

The goal of most well-constructed catch clauses should be to resolve the exceptional condition and then continue on as if the error had never happened.

```
     import java.util.Random;
     class HandleError
     {
          public static void main(String args[])
          {
               int a=0, b=0, c=0;
               Random r = new Random();

               for(int i=0; i<10; i++)
               {
                    try
                    {
                         b = r.nextInt();
                         c = r.nextInt();
                         a = 12345 / (b/c);
                    } catch (ArithmeticException e)
                    {
                         System.out.println("Division by zero.");
                         a = 0;
                    }
                    System.out.println("a: " + a);
               }
          }
     }
```

The output of above program is not predictable exactly, as we are generating random numbers. But, the loop will execute 10 times. In each iteration, two random numbers (b and c) will be generated. When their division results in zero, then exception will be caught. Even after exception, loop will continue to execute.

Displaying a Description of an Exception: We can display this description in a println() statement by simply passing the exception as an argument. This is possible because Throwable overrides the toString() method (defined by Object) so that it returns a string containing a description of the exception.

```
catch (ArithmeticException e)
{
     System.out.println("Exception: " + e);
     a = 0;
}
```

Now, whenever exception occurs, the output will be –

```
     Exception: java.lang.ArithmeticException: / by zero
```

4.3.5 Multiple Catch Claues

In some cases, more than one exception could be raised by a single piece of code. To handle this type of situation, you can specify two or more catch clauses, each catching a different type of exception. When an exception is thrown, each catch statement is inspected in order, and the first one whose type matches that of the exception is executed. After one catch statement executes, the others are bypassed, and execution continues after the try/catch block.

```
class MultiCatch
{
     public static void main(String args[])
     {
          try
          {
               int a = args.length;
               System.out.println("a = " + a);
               int b = 42 / a;
               int c[] = { 1 };
               c[42] = 99;
          } catch(ArithmeticException e)
          {
               System.out.println("Divide by 0: " + e);
          } catch(ArrayIndexOutOfBoundsException e)
          {
               System.out.println("Array index oob: " + e);
          }
               System.out.println("After try/catch blocks.");
     }
}
```

Here is the output generated by running it both ways:

```
C:\>java MultiCatch
a = 0
Divide by 0: java.lang.ArithmeticException: / by zero
After try/catch blocks.

C:\>java MultiCatch TestArg
a = 1
Array index oob: java.lang.ArrayIndexOutOfBoundsException:42
After try/catch blocks.
```

While using multiple *catch* blocks, we should give the exception types in a hierarchy of subclass to superclass. Because, *catch* statement that uses a superclass will catch all exceptions of its own type plus all that of its subclasses. Hence, the subclass exception given after superclass exception is never caught and is a *unreachable code,* that is an *error* in Java.

```
class SuperSubCatch
{
      public static void main(String args[])
      {
            try
            {
                  int a = 0;
                  int b = 42 / a;
            } catch(Exception e)
            {
                  System.out.println("Generic Exception catch.");
            }
            catch(ArithmeticException e)     // ERROR - unreachable
            {
                  System.out.println("This is never reached.");
            }
      }
}
```

The above program generates error “Unreachable Code”, because ArithmeticException is a subclass of Exception.

4.3.6 Nested try Statements

The try statement can be nested. That is, a try statement can be inside the block of another try. Each time a try statement is entered, the context of that exception is pushed on the stack. If an inner try statement does not have a catch handler for a particular exception, the stack is unwound and the next try statement’s catch handlers are inspected for a match. This continues until one of the catch statements succeeds, or until all the nested try statements are exhausted. If no catch statement matches, then the Java run-time system will handle the exception.

```
class NestTry
{
      public static void main(String args[])
      {
            try
            {
                int a = args.length;
                int b = 42 / a;

                System.out.println("a = " + a);

                try
                {
                 if(a==1)
                        a = a/(a-a);
```

```
                     if(a==2)
                     {
                            int c[] = { 1 };
                            c[10] = 99;
                     }
                    }catch(ArrayIndexOutOfBoundsException e)
                    {
                     System.out.println("Array index out-of-bounds: " + e);
                    }
                  }catch(ArithmeticException e)
                  {
                     System.out.println("Divide by 0: " + e);
                  }
          }
     }
```

When a method is enclosed within a try block, and a method itself contains a try block, it is considered to be a nested try block.

```
class MethNestTry
{     static void nesttry(int a)
      {     try
            {
                  if(a==1)
                        a = a/(a-a);
                  if(a==2)
                  {
                        int c[] = { 1 };
                        c[42] = 99;
                  }
            }catch(ArrayIndexOutOfBoundsException e)
             {
                  System.out.println("Array index out-of-bounds: " + e);
             }
      }

      public static void main(String args[])
      {
            try
            {
                  int a = args.length;
                  int b = 42 / a;
                  System.out.println("a = " + a);
                  nesttry(a);
            } catch(ArithmeticException e)
              {
                  System.out.println("Divide by 0: " + e);
              }
      }
}
```

4.3.7 throw

Till now, we have seen catching the exceptions that are thrown by the Java run-time system. It is possible for your program to throw an exception explicitly, using the ***throw*** statement. The general form of **throw** is shown here:

```
throw ThrowableInstance;
```

Here, *ThrowableInstance* must be an object of type **Throwable** or a subclass of **Throwable**. Primitive types, such as **int** or **char**, as well as non-**Throwable** classes, such as **String** and **Object**, cannot be used as exceptions.

There are two ways you can obtain a **Throwable** object:

- using a parameter in a **catch** clause, or
- creating one with the **new** operator.

```
class ThrowDemo
{
     static void demoproc()
     {
          try
          {
               throw new NullPointerException("demo");
          } catch(NullPointerException e)
          {
               System.out.println("Caught inside demoproc: " + e);

          }
     }

     public static void main(String args[])
     {
          demoproc();
     }
}
```

Here, **new** is used to construct an instance of **NullPointerException**. Many of Java's built-in run-time exceptions have at least two constructors:

- one with no parameter and
- one that takes a string parameter

When the second form is used, the argument specifies a string that describes the exception. This string is displayed when the object is used as an argument to **print()** or **println()**. It can also be obtained by a call to **getMessage()**, which is defined by **Throwable**.

4.3.8 throws

If a method is capable of causing an exception that it does not handle, it must specify this behavior so that callers of the method can guard themselves against that exception. You do this by including a **throws** clause in the method's declaration. A **throws** clause lists the types of exceptions that a method might throw. This is necessary for all exceptions, except those of type **Error** or **RuntimeException**, or any of their subclasses. All other exceptions that a method can throw must be declared in the **throws** clause. If they are not, a compile-time error will result.

The general form of a method declaration that includes a **throws** clause:

```
	type method-name(parameter-list) throws exception-list
	{
		// body of method
	}
```

Here, *exception-list* is a comma-separated list of the exceptions that a method can throw.

```
	class ThrowsDemo
	{
		static void throwOne() throws IllegalAccessException
		{
			System.out.println("Inside throwOne.");
			throw new IllegalAccessException("demo");
		}
		public static void main(String args[])
		{
			try
			{
				throwOne();
			} catch (IllegalAccessException e)
			{
				System.out.println("Caught " + e);
			}
		}
	}
```

4.3.9 finally

When exceptions are thrown, execution in a method takes a rather abrupt, nonlinear path that alters the normal flow through the method. Sometimes it is even possible for an exception to cause the method to return prematurely. This could be a problem in some methods. For example, if a method opens a file upon entry and closes it upon exit, then you will not want the code that closes the file to be bypassed by the exception-handling mechanism. The **finally** keyword is designed to address such situations.

The **finally** clause creates a block of code that will be executed after a **try**/**catch** block has completed and before the next code of **try/catch** block. The **finally** block will execute whether or not an exception is thrown. If an exception is thrown, the **finally** block will execute even if no **catch** statement matches the exception. Any time a method is about to return to the caller from inside a **try/catch** block, via an uncaught exception or an explicit return statement, the **finally** clause is also executed just before the method returns. The **finally** clause is optional. However, each **try** statement requires at least one **catch** or a **finally** clause.

```
class FinallyDemo
{
	static void procA()
	{
		try
		{
			System.out.println("inside procA");
			throw new RuntimeException("demo");
		} finally
		 {
			System.out.println("procA's finally");
		  }
	}

	static void procB()
	{
		try
		{
			System.out.println("inside procB");
			return;
		} finally
		 {
			System.out.println("procB's finally");
		  }
	}

	static void procC()
	{	try
		{
			System.out.println("inside procC");
		} finally
		{
			System.out.println("procC's finally");
		}
	}

	public static void main(String args[])
	{
		try
		{
			procA();
		} catch (Exception e)
		{
			System.out.println("Exception caught");
		}
		procB();
		procC();
	}
}
```

Output:

```
	inside procA
	procA’s finally
	Exception caught
	inside procB
	procB’s finally
	inside procC
	procC’s finally
```

4.3.10 Java's Built-in Exceptions

Inside the standard package **java.lang**, Java defines several exception classes. The most general of these exceptions are subclasses of the standard type **RuntimeException**. These exceptions need not be included in any method's **throws** list. Such exceptions are called as ***unchecked exceptions*** because the compiler does not check to see if a method handles or throws these exceptions. Java.lang defines few ***checked exceptions*** which needs to be listed out by a method using ***throws*** list if that method generate one of these exceptions and does not handle it itself. Java defines several other types of exceptions that relate to its various class libraries.

Table: Java's Unchecked Exceptions

Exception	**Meaning**
ArithmeticException	Arithmetic error, such as divide-by-zero.
ArrayIndexOutOfBoundsException	Array index is out-of-bounds.
ArrayStoreException	Assignment to an array element of an incompatible type.
ClassCastException	Invalid cast.
EnumConstantNotPresentException	An attempt is made to use an undefined enumeration value.
IllegalArgumentException	Illegal argument used to invoke a method.
IllegalMonitorStateException	Illegal monitor operation, such as waiting on an unlocked thread.
IllegalStateException	Environment or application is in incorrect state.
IllegalThreadStateException	Requested operation not compatible with current thread state.
IndexOutOfBoundsException	Some type of index is out-of-bounds.
NegativeArraySizeException	Array created with a negative size.
NullPointerException	Invalid use of a null reference.
NumberFormatException	Invalid conversion of a string to a numeric format.
SecurityException	Attempt to violate security.
StringIndexOutOfBounds	Attempt to index outside the bounds of a string.
TypeNotPresentException	Type not found.
UnsupportedOperationException	An unsupported operation was encountered.

Table: Java's Checked Exceptions

Exception	Meaning
ClassNotFoundException	Class not found.
CloneNotSupportedException	Attempt to clone an object that does not implement the Cloneable interface.
IllegalAccessException	Access to a class is denied.
InstantiationException	Attempt to create an object of an abstract class or interface.
InterruptedException	One thread has been interrupted by another thread.
NoSuchFieldException	A requested field does not exist.
NoSuchMethodException	A requested method does not exist.

4.3.11 Creating your own Exception Subclasses

Although Java's built-in exceptions handle most common errors, sometimes we may want to create our own exception types to handle situations specific to our applications. This is achieved by defining a subclass of ***Exception*** class. Your subclasses don't need to actually implement anything—it is their existence in the type system that allows you to use them as exceptions. The **Exception** class does not define any methods of its own. It inherits those methods provided by **Throwable**. Thus, all exceptions, including those that you create, have the methods defined by **Throwable** available to them.

Method	**Description**
Throwable fillInStackTrace( )	Returns a Throwable object that contains a completed stack trace. This object can be re-thrown.
Throwable getCause( )	Returns the exception that underlies the current exception. If there is no underlying exception, null is returned.
String getLocalizedMessage( )	Returns a localized description of the exception.
String getMessage()	Returns a description of the exception.
StackTraceElement[] getStackTrace()	Returns an array that contains the stack trace, one element at a time, as an array of StackTraceElement. The method at the top of the stack is the last method called before the exception was thrown. This method is found in the first element of the array. The StackTraceElement class gives your program access to information about each element in the trace, such as its method name.
Throwable initCause(Throwable causeExc)	Associates causeExc with the invoking exception as a cause of the invoking exception. Returns a reference to the exception.
void printStackTrace( )	Displays the stack trace.

void printStackTrace(PrintStream stream)	Sends the stack trace to the specified stream.
void printStackTrace(PrintWriter stream)	Sends the stack trace to the specified stream.
void setStackTrace(StackTraceElement elements[ ])	Sets the stack trace to the elements passed in elements. This method is for specialized applications, not normal use.
String toString( )	Returns a String object containing a description of the exception. This method is called by println( ) when outputting a Throwable object.

We may wish to override one or more of these methods in exception classes that we create. Two of the constructors of **Exception** are:

Exception( )
Exception(String *msg*)

Though specifying a description when an exception is created is often useful, sometimes it is better to override **toString( )**. The version of **toString( )** defined by **Throwable** (and inherited by **Exception**) first displays the name of the exception followed by a colon, which is then followed by your description. By overriding **toString( )**, you can prevent the exception name and colon from being displayed. This makes for a cleaner output, which is desirable in some cases.

```
class MyException extends Exception
{
      int marks;

      MyException (int m)
      {
         marks=m;
      }

      public String toString()
      {
         return "MyException: Marks cannot be Negative";
      }
}
class CustExceptionDemo
{
      static void test(int m) throws MyException
      {
         System.out.println("Called test(): "+m);
         if(m<0)
            throw new MyException(m);

          System.out.println("Normal exit");
      }
```

```
        public static void main(String args[])
        {
           try{
             test(45);
             test(-2);
           }
           catch (MyException e)
           {
             System.out.println("Caught " + e);
            }
        }
    }
```

4.3.12 Chained Exceptions

The concept of *chained exception* allows you to associate another exception with an exception. This second exception describes the cause of the first exception. For example, imagine a situation in which a method throws an ArithmeticException because of an attempt to divide by zero. However, the actual cause of the problem was that an I/O error occurred, which caused the divisor to be set improperly. Although the method must certainly throw an ArithmeticException, since that is the error that occurred, you might also want to let the calling code know that the underlying cause was an I/O error. Chained exceptions let you handle this, and any other situation in which layers of exceptions exist.

To allow chained exceptions, two constructors and two methods were added to **Throwable**. The constructors are shown here:

- Throwable(Throwable *causeExc*)
- Throwable(String *msg*, Throwable *causeExc*)

In the first form, *causeExc* is the exception that causes the current exception. That is, *causeExc* is the underlying reason that an exception occurred. The second form allows you to specify a description at the same time that you specify a cause exception. These two constructors have also been added to the **Error**, **Exception**, and **RuntimeException** classes.

Chained exceptions can be carried on to whatever depth is necessary. Thus, the cause exception can, itself, have a cause. Be aware that overly long chains of exceptions may indicate poor design. Chained exceptions are not something that every program will need. However, in cases in which knowledge of an underlying cause is useful, they offer an elegant solution.

4.3.13 Using Exceptions

Exception handling provides a powerful mechanism for controlling complex programs that have many dynamic run-time characteristics. It is important to think of try, throw, and catch as clean ways to handle errors and unusual boundary conditions in your program's logic. Unlike some other languages in which error return codes are used to indicate failure, Java uses exceptions. Thus, when a method can fail, have it throw an exception. This is a cleaner way to handle failure modes.

Note that Java's exception-handling statements should not be considered a general mechanism for nonlocal branching. If you do so, it will only confuse your code and make it hard to maintain.

Question Bank:

1. What do you mean by a package? How do you use it in a Java program? Explain with a program.
2. How do you import a package? Explain.
3. Write a note on access protection in Java.
4. Define an interface. Explain how to define and implement an interface with an example.
5. Differentiate abstract base class and an interface.
6. How do you define variables inside interface? List out the the characteristics of such variables.
7. Define an exception. What are the key terms used in exception handling? Explain.
8. Demonstrate working of nest try block with an example.
9. Write a program which contains one method which will throw IllegalAccessException and use proper exception handles so that exception should be printed.
10. Write a note on:
    a. Java's built-in exception
    b. Uncaught Exceptions
11. How do you create your own exception class? Explain with a program.